THE VECTORIZATION OF A RAY TRACING PROGRAM FOR IMAGE GENERATION

DAVID J. PLUNKETT,
JOSEPH M. CYCHOSZ
AND
MICHAEL J. BAILEY

PURDUE UNIVERSITY CADLAB

WEST LAFAYETTE, INDIANA

THE VECTORIZATION OF A RAY TRACING PROGRAM FOR IMAGE GENERATION

David J. Plunkett[1]

Joseph M. Cychosz

Michael J. Bailey

Purdue University CADLAB

ABSTRACT

Ray tracing is a widely used method for producing realistic computer-generated images. Ray tracing involves firing an imaginary ray from a view point, through a point on an image plane, into a three dimensional scene. The intersection of the ray with the objects in the scene determines what is visible at that point on the image plane. This process must be repeated many times, once for each point (commonly called a pixel) in the image plane. A typical image contains more than a million pixels making this process computationally expensive. A traditional ray tracing program processes one ray at a time. In such a serial approach, as much as ninety percent of the execution time is spent computing the intersection of a ray with the surfaces in the scene. With the CYBER 205, many rays can be intersected with all the bodies in the scene with a single series of vector operations. Vectorization of this intersection process results in large decreases in computation time.

The CADLAB's interest in ray tracing stems from the need to produce realistic images of mechanical parts. A high quality image of a part during the design process can increase the productivity of the designer by helping him visualize the results of his work. To be useful in the design process, these images must be produced in a reasonable amount of time. This discussion will explain how the ray tracing process was vectorized and gives examples of the images obtained.

1. Authors' Address:
CADLAB, Potter Engineering Center
Purdue University
West Lafayette, IN 47907
(317) 494-5944

GEOMETRIC MODELING AND MECHANICAL DESIGN

In mechanical design, there are two broad reasons for using the computer: (1) predict behavior, and (2) visualize. Behavior that needs to be predicted includes every test that one would normally perform if given a physical prototype of the design: weight, center of gravity, strength, movement, clearances, etc. This is why a computer model of a part is often referred to as a "virtual prototype." Visualization is, in effect, another form of behavior prediction. In this case, knowing the actual appearance of a proposed design is a valuable aid in conceptualizing.

In order to feed information into visualization and analysis routines, a *geometric model* of the design must first be created. In the early days of computer aided engineering, a wireframe database was used to model the part shape. This was deemed inadequate, because the wireframe could only model a part's *edges*, not its *solid volume*.

One of the methods by which we model part shapes in the CADLAB is with a newer technique called *Solid Modeling*. A solid modeling database has sufficient geometric information to completely and unambiguously define the shape of a three dimensional object. One method of building a solid model database is with a technique called *Constructive Solid Geometry,* or CSG. A CSG geometric creation sequence is characterized by applying boolean operators (union, difference, intersection) to groups of primitive shapes (boxes, cylinders, cones, etc). Complex designs may be created in this manner, with the results being sufficient to drive visualization and other analyses. The remainder of this report will discuss the use of the CYBER 205 to produce image information in order to view an object constructed using CSG operations.

INTERSECTIONS OF RAYS WITH A PRIMITIVE

One nice side effect of using a CSG representation is that the resulting object can easily be displayed using ray tracing. Ray tracing involves firing an imaginary ray from a view point, through a point on an image plane, into a three dimensional scene. It is not mathematically feasible to determine the visible surface of an entire *CSG object* in a single computation. However, it is fairly easy to determine the intersection of a ray with each of the ***individual primitives*** which make up a CSG object. Then, a little more calculation produces the point along that ray which is visible. If one ray is fired through every pixel in the image plane, an image of the object is obtained (see Figure 1).

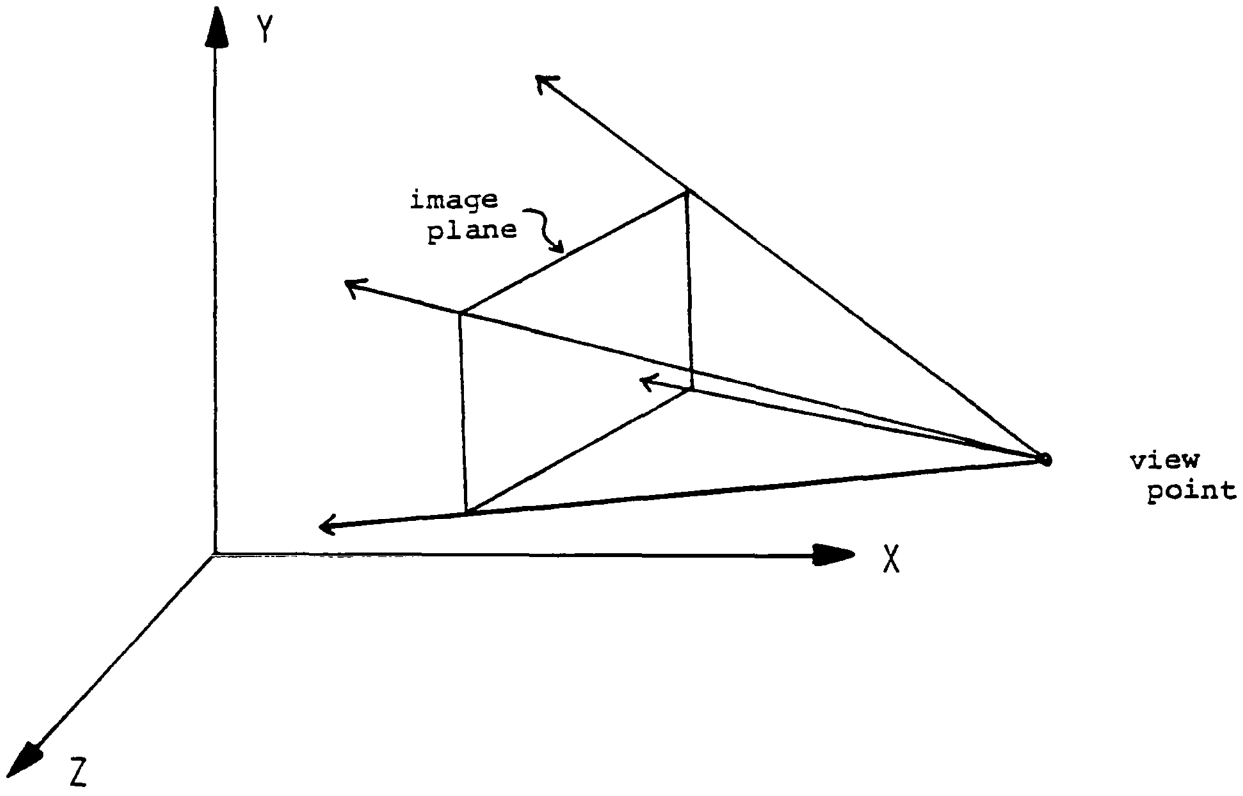

Figure 1. The Image Environment.

The typical (serial) ray tracing program must:

- Intersect all primitives in the scene with one ray.
- Traverse the CSG database to determine which primitive intersection is the visible surface for that ray.
- Determine the surface intensity using the surface relationship between the surface normal, the eye position, and the position(s) of the light source(s).

This is the visible surface algorithm. It is repeated at every picture element (pixel) in the image plane.

The intersection of the ray with the primitives is by far the most time consuming part of the visible surface algorithm. However, it is also the easiest part of the algorithm to vectorize. Instead of just finding the intersection of one ray with a primitive, a queue of rays is built (serially as in a traditional ray tracing program). Then the intersections of each primitive with every ray in the queue is found in a series of vector operations. Table 1 gives computation times for 100,000 rays intersecting a sphere and a cylinder primitive. For the vector results in this table, a queue length of 2000 rays was used.

FINDING A RAY'S VISIBLE SURFACE

The above timings are only for the lowest level in the visible surface algorithm. After all the intersections are found, the CSG database must still be traversed to determine which primitive intersection is the visible surface for that ray. This constrains the length of the ray queue, since it implies that all the ray intersection information must be stored (after the intersection calculation) and then retrieved (for the visible surface calculation). If the ray queue is too long, the time spent page faulting will be enormous. For this reason, the ray queue in our application is

TABLE 1.

CPU Times[2]

Primitive	Cyber 205 Scalar	Cyber 205 Vector	Cyber 720
sphere	.944	.0279	13.1
cylinder	2.729	.1614	51.48
steiner	11.157	1.047	216.0

Speedup[3]

Primitive	$S_{205\ scalar}^{205\ vector}$	$S_{720}^{205\ vector}$
sphere	33.81	469
cylinder	16.91	318
steiner	10.67	206

2 CPU times are in seconds

3 Speedup $= S_{P_2}^{P_1} = \dfrac{\text{CPU time } P_2}{\text{CPU time } P_1}$

approximately 2000 rays. The visible surface algorithm has not yet been vectorized. However, it is apparent that at least parts of this process are vectorizable.

SPECIAL EFFECTS

One of the reasons ray tracing has been so widely accepted is that it can show very realistic image synthesis effects. Shadows are perhaps the easiest extension to the algorithms described above. To determine if a visible surface is in a shadow, one ray must be fired toward each light source from the visible surface. If this ray hits a solid object before it encounters the light source, the visible surface is in a shadow. Reflection can be shown by spawning another ray from each surface such that the angle of reflection equals the angle of incidence. Transparency and refraction can be modeled if a refraction ray is spawned after a hit on a solid, transparent object. What should be clear from these special effects is that the extra rays to be fired do not come in a predictable, vectorizable progression. However, after a serial section of code has determined that another ray must be fired, this ray can be placed in the queue and intersected using vector code when the queue is full.

SURFACE PATCHES

Surface patches are used in computer aided design to sculpt the surface of a part that would be difficult or impossible to model using conventional primitives such as cylinders and boxes. Hence, surface patches play an important role in the design process of parts such as air foils and car bodies. At the CADLAB we are currently investigating the uses of Steiner surfaces as a sculpting device. Ray tracing is then used to visualize the resulting sculpted surface.

A Steiner surface is a bi-quadratic surface. This means that computing the intersection of a ray with a Steiner surface requires the solving of a quartic equation. Approximately 65 precent of the computation time for this intersection calculation involves the solving of the quartic equation while the rest is attributed to the determination of the coefficients for the quartic equation. The determination of the polynomial coefficients is a straight forward process and is easily vectorized. Vectorizing the process by which a queue of rays may be intersected with a Steiner surface requires the vectorization of the root solver used for solving the quartic. For our application we are only interested in the first positive real root closest to zero. Table 1 shows the results of vectorizing the Steiner intersection process.

To determine the roots of the quartic polynomial the slope and curvature functions (i.e. the first and second derivatives) are examined to determine the intervals over which a possible solution exists. Modified Regula Falsi is then used to determine the roots within these intervals. Once a root is found it is evaluated to see if the root is acceptable.

The vectorized version of the root solver finds the roots of a series of quartic polynomials, each polynomial corresponding to a ray in the ray queue. The roots for all the polynomials must be found before the process can complete. Unlike the scalar version, it is most likely that all four roots will have to be determined and evaluated as it is likely that at least one ray will not intersect the surface. This process is sped up by ensuring that a sign change does not occur before using the Falsi method to determine subsequent roots once an acceptable root has been found for a particular polynomial. Gather-scatters are then used to compress the vectors used during these iterative processes. Convergence occurs when all of the roots being found converge within the specified tolerance.

The quartic root solver can be used for a variety of applications. One extension to the ray tracing program will be the inclusion of tori and other elliptical surfaces as primitives. These primitives will also require solving a fourth order equation to determine the intersection of a ray with their surface.

OTHER APPLICATIONS

Another application of ray tracing at Purdue is radiant heat transfer analysis of finned Tubes [MAXW83].4 Rays are fired to determine the radiation shape factor of one or more finned tubes. Unlike the visualization of a CSG object, maximum length vector operations may be used since it is only of interest knowing that the ray strikes the tube and not where on the tube. The computational requirements of this application have been reduced from 600 seconds on a CDC 6600 down to 3 seconds on the CYBER 205.

CONCLUSION

Ray tracing is, in general, a *parallel* algorithm. This paper examined how the parallel algorithm can be modified for use on a vector computer. In design work, the speed with which results are available is often critical. Vectorization of ray tracing programs promises shorter execution times. This will benefit not only visualization, but also such diverse areas as heat transfer, mass properties analysis, and nuclear engineering.

4 [MAXW83] Maxwell, G.M., "Mathematical Modelling of a Gas Fired Swimming Pool Water Heater", Ph.D. Thesis, Purdue University, in preparation.